Seth Cram

ECE341

Lab8 Prelab

11/02/2021

**Prelab 8: I2C and EEPROM Communication**

Goal:

To delve into synchronous communication through using a master-slave multi-drop network communication protocol called I2C.

Background Information:

We’ll use the I2C protocol to communicate with the serial EEPROM. I2C stands for Inter-Integrated Circuit protocol. I2C is also a serial protocol. We’ll need a synchronizing clock as a separate signal for I2C. I2C is supported by hardware in a large number of microprocessors. I2C is half-duplex since its data line is bi-directional, but can’t send data in both directions at the same time. For this lab, the PIC32 will act as the master as the EEPROM acts as the slave, so we’ll only have one slave but we could theoretically add more onto the I2C bus. The master doesn’t have to stay the same the entire time and can change, but it’ll only be the PIC32 for our purposes.

Each slave has a unique device ID number, addressable through the 7-bit address sent from the master. An I2C Frame consists of a Start condition, four bit control code, three bit chip select address, a read/write, and an acknowledge bit. The transfer ends with a Stop condition from the Master. The acknowledged bit must be set low by the receiving device after a data transfer. Otherwise, an error occurred with the transfer. The read/write line indicates the direction of the next data transfer. After an I2C frame, we can read or write successive bytes.

The I2C frame occurs on the SDA (Serial Data Line), with each bit separated by the SCL (Serial Clock Line) going low. SDA goes high to low for a start and low to high for a stop, all while SCL is high. Otherwise, SDA can't change while SCL is high. Which makes our communication synchronous. The most significant bit is sent first.

The I2C controller will help us satisfy timing constraints. The slowest device in the I2C network will dictate the maximum data transfer rate. When the memory chip encounters an error, it won’t be readily apparent. We might have to cycle the power. We’ll only be able to tell from the lack of an acknowledged bit when writing to it.

The EEPROM (electrically erasable programmable read only memory) is nonvolatile, so it still saves data even when powered off. It’s capable of storing 256 Kbits of data, or 32,768 bytes. Writes are slow because the data we write is first stored in the Page Latches before being placed into the EEPROM array. Reads are fast because they don’t require the middle man. The EEPROM array is where the addresses of the memory chip are located.

We will poll the acknowledge bit to know when the receiver is ready for a data transfer. This will be done by repeatedly writing to the EEPROM and seeing whether the acknowledge is set low. To end a data transfer, the receiver needs to not set the acknowledge bit low. This is referred to as “NACK” or no acknowledge. Otherwise, data bytes can be read/written back-to-back. The source of a NACK could be the device is busy, isn’t functional, or an incorrect device address was sent.

We’ll create an EEPROM device driver to write and read a reasonable number of bytes to/from the EEPROM in succession. For this, we’ll have to take pages into account. The EEPROM’s page latch is 64 bytes. When writing information into the EEPROM, we’ll have to take into account that it stores this information in its page latch, and we can’t write to it when it commits this information from the page latches into its actual memory array. This commitment process happens at the end of each page.

Plan:

I’ll base my design off of the code supplied to us during the first week of this lab. With this method, the polling of the EEPROM to check if it's ready for another data transfer was pretty much already done. We’d only have to add a counting variable to exit the loop when too many writes have been attempted.

Moving onto the EEPROM read function, I’ll do some error checking before reading any data. First, I’ll verify the memory address is within range by comparing its specified page number to the maximum page numbers addressable. Next, I’ll make sure length is positive, and the passed in array isn’t NULL. Finally, I’ll use the code supplied for a read operation. Instead of writing a set number of bytes, I’ll individually write out the control bytes and memory address, then start looping for however many bytes I’ll need to read-in. Some error checking done in the loop include the current array position being inaccessible, the page number becoming too large, and the length not reaching zero so we can trigger an acknowledgement. We’ll issue a no-acknowledge and stop if an error occurs, or if we’ve read in all the requested bytes.

For the EEPROM write function, we’ll have to consider page numbers more integrally. The same error checking as for the read function should be implemented. Following the same structure as the reading function, we’ll differentiate by not changing the bus direction, and allowing the EEPROM time to commit the data passed into the page latches when a different page is crossed into. When considering pages we’ll need to use a counting variable initialized to the memory address and increment it everytime we write data. Unfortunately, there’s no way for us to check the offset within a page directly, so we keep track of it manually.

During the test file, we’ll have to initialize a variety of test variables to both pass into the read and write functions, and keep track of the error codes they return. First, we’ll fill our test array with the same character using a for loop. Then, we’ll use the above write function to commit the entire array to the EEPROM’s memory array. We’ll have to pass the length of our array, our array, the desired memory address to write to, and the EEPROM’s slave address. In return, we’ll get the write error code. We also pass in the same parameters to the read function, but a different array that’s empty and receives the read error code. Finally, we’ll loop through each individual array entry and verify that the data received is the same as the data written out to the EEPROM. If so, we’ll output a success to the LCD, otherwise we’ll output a failure. We could also print the error codes after some delay, if desired.



